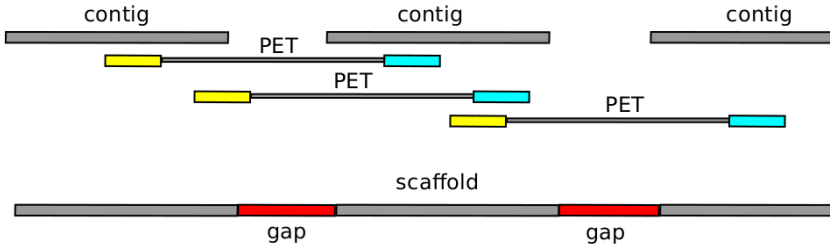
[MBI.A] Asembler DNA, sparowane końce - dokumentacja końcowa

Michał Aniserowicz, Jakub Turek

Opis problemu

Zadanie polega na implementacji aplikacji, która umożliwia tworzenie *scaffoldów* na podstawie dostarczonych zbiorów *contigów* oraz sekwencji PET.



Założenia

W ogólnym przypadku rekonstrukcja sekwencji *contigów* nie jest możliwa. Z tego względu, na potrzeby projektu przyjęto następujące założenia:

- Początki i końce łańcuchów PET to sekwencje unikalne. Wystąpienie takiej sekwencji w jednym z *contigów* oznacza, że jest to odpowiednio początek lub koniec sekwencji PET.
- Badane są wyłącznie takie permutacje *contigów*, dla których wystąpienie początku sekwencji PET implikuje przynajmniej częściowe wystąpienie jej końca w dalszej części łańcucha. Innymi słowy początek lub koniec sekwencji PET nie może w całości wystąpić w przerwie (*gap*) *scaffoldu*.
  - Wyjątkiem od tej reguły jest początek i koniec *scaffoldu*, gdzie mogą występować, odpowiednio, niesparowane końce lub początki sekwencji PET.
- Sekwencje należące do różnych par sparowanych końców mogą częściowo zachodzić na siebie.

Algorytm

Do rozwiązania zadania użyty został algorytm typu brute-force działający na wstępnie odfiltrowanym zbiorze kombinacji contigów. Algorytm przebiega według następującego schematu:

1. Wyznaczane są wszystkie permutacje zadanego zbioru *contigów*.
2. Permutacje zostają wstępnie sprawdzone:
   - następuje próba znalezienia sekwecji PET, której początek i koniec w całości zawiera się w dowolnej parze *contigów*:
     - jeśli taka sekwencja nie zostanie znaleziona, wszyskie permutacje zostają uwzględnione w kolejnych krokach algorytmu;
   - każda permutacja jest sprawdzana pod względem kolejności wystąpienia *contigów* zawierających znalezioną sekwencję:
     - jeśli *contig* zawierający początek sekwencji znajduje się przed *contigiem* zawierającym jej koniec, to dana permutacja zostaje uwzględniona w kolejnych krokach algorytmu,
     - w przeciwnym wypadku, permutacja zostaje odrzucona.
3. Na podstawie każdej z zaakceptowanych permutacji budowany jest *scaffold*:
   - dla każdej sekwencji PET:
     (a) następuje próba znalezienia *contiga* zawierającego (w całości lub częściowo[1]) początek sekwencji,
     (b) przeglądane są *contigi* występujące po znalezionym, w celu odnalezienia *contiga* zawierającego koniec sekwecji,
     (c) jeśli odległość pomiędzy znalezioną parą *conitgów* dopuszcza istnienie pomiędzy nimi sekwencji PET, to pomiędzy te *contigi* wstawiany jest odpowiedniej długości *gap*,
     (d) jeśli w którymkolwiek z powyższych kroków nastąpiło niepowodzenie (odpowiedni *conitg* nie został odnaleziony lub sprawdzenie odległości dało wynik negatywny), to wykonywane jest sprawdzenie, czy pierwszy *contig* zawiera koniec sekwecji lub czy ostatni *conitg* zawiera jej początek - taka sytuacja uznawana jest za poprawną,
     (e) aktualizowany jest ranking R, określający liczbę pokrywających się zasad początku lub końca sekwencji PET z odnalezionymi *contigami*.
   - jeżeli $R > 0$, to dany *scaffold* dodawany jest listy wynikowej algorytmu.
4. Uzyskana lista *scaffoldów* sortowana jest według malejących wartości R.

[1] Częściowe pokrycie sekwencji PET występuje, gdy *contig* zaczyna się końcem fragmentu (tzn. początku lub końca) danej sekwencji lub kończy początkiem fragmentu tej sekwecji. Minimalny akceptowalny stosunek części pokrytej do całości fragmentu określony jest w pliku konfiguracyjnym aplikacji.

Implementacja

Projekt został zaimplementowany w języku C# (platforma .NET). Testy jednostkowe zostały napisane w oparciu o platformę NUnit[2], z użyciem biblioteki Rhino Mocks[3].

Aplikacja posiada interfejs okienkowy stworzony w technologii WPF, umożliwiający odczyt danych wejściowych z/zapis danych wyjściowych do pliku. Na dane wejściowe składają się:

- Opis *conitgów* w postaci łańcuchów tekstowych oddzielonych znakami nowej linii.

```
ACAGCTTA
CCGGGTAC
TACAGCTT
```

- Opis sekwencji PET w postaci dwóch sekwencji (początek, koniec) oraz długości łańcucha. Dane w sekwencji oddzielone przecinkami, natomiast kolejne PET'y oddzielone znakami nowej linii.

```
GATC,CCAT,100
GGCT,AGAA,1500
```

Dane wyjściowe to uporządkowana sekwencja *contigów* oddzielonych znakami spacji reprezentującymi długość przerwy (*gap*).

```
CCGGGTAC       TACAGCTT  ACAGCTTA
```

Przykład

- Plik wejściowy:

```
ACAGCTTA
CCGGGTAC
TACAGCAA

CCG,TACA,15
GCA,GCT,13
```

- Plik wyjściowy:

```
CCGGGTAC   TACAGCAA   ACAGCTTA
```

[2]http://www.nunit.org/
[3]http://www.ayende.com/wiki/Rhino+Mocks.ashx

Testowanie

Program wykonany w ramach projektu tworzony był z wykorzystaniem metodyki Test Driven Development[4], czego skutkiem jest znaczne pokrycie testami jednostkowymi - testów jest ok. 70. Przetestowane zostały m.in. następujące przypadki:

- odrzucenie niepoprawnych permutacji *contigów* (patrz krok 2 algorytmu),
- odnalezienie całkowitego pokrycia fragmentów sekwencji PET przez sąsiadujące *contigi* (kroki 3a, 3b),
- odnalezienie całkowitego pokrycie fragmentów sekwencji PET przez *contigi* oddalone od siebie (kroki 3a, 3b),
- odnalezienie wystąpienia końca fragmentu sekwencji PET w początku *contigu* (krok 3a / 3b),
- odnalezienie wystąpienia początku fragmentu sekwencji PET w końcu *contigu* (krok 3a / 3b),
- odnalezienie całkowitego pokrycia pomimo występującego we wcześniejszym *contigu* pokrycia częściowego (kroki 3a / 3b),
- wstawienie *gapów* pomiędzy dwa *contigi* całkowicie pokrywające fragmenty sekwencji PET (krok 3c),
- wstawienie *gapów* pomiędzy dwa *contigi* częściowo pokrywające fragmenty sekwencji PET (krok 3c),
- weryfikacja odległości pomiędzy *conitgami* (krok 3c),
- odnalezienie niesparowanego końca sekwencji PET w pierszym *contigu* (krok 3d),
- odnalezienie niesparowanego początku sekwencji PET w ostatnim *contigu* (krok 3d),
- obliczanie rankingu R *scaffoldu* (krok 3e),
- odrzucenie *scaffoldów* o zerowej wartości rankingu R (krok 3),
- sortowanie *scaffoldów* według wartości rankingu R (krok 4).

Ponadto dokonano wielokrotnych "ręcznych" testów z wykorzystaniem przygotowanych danych wejściowych. Wynik wszystkich testów był pozytywny.

[4] http://pl.wikipedia.org/wiki/Test-driven_development

Wnioski

Proces budowania poprawnego *scaffoldu* wydaje się być prostą i wydajną sekwencją operacji na ciągach znaków. Biorąc jendak pod uwagę ilość kombinacji, które należy sprawdzić ($n!$, gdzie n to liczność zadanego zbioru *contigów*), nie dziwi fakt, że algorytmy tego typu są wymagające zarówno pamięciowo jak i obliczeniowo.

Potencjalnym usprawnieniem działania algorytmu może być zastosowanie lepszej heurystyki podczas wstępnego filtrowania kombinacji *contigów*. Obecna implementacja odrzuca tylko około połowę kombinacji, co dla zestawu dziesięciu *conitgów* pozostawia do dalszego przetworzenia ok. 1,8 mln kombinacji.